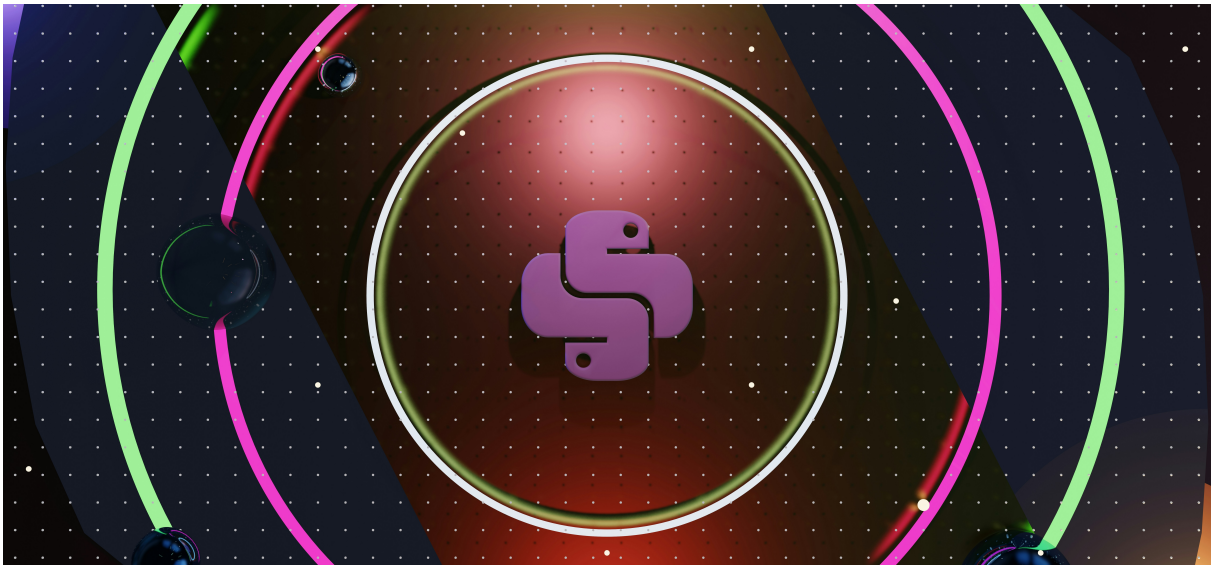

Django.

▼ Models

Django models form the foundation of how data is structured and stored in a Django application. They map Python object to relational database tables, allowing to interact with databases using Python code rather than writing SQL directly.

▼ Introduction

A Django model is a Python class that maps to a database table. Each attribute of the model class represents a column in a database. Django handles communication between Python code and the database, allowing to perform CRUD operations.

▼ Basic example of a Django model

```
from django.db import models
class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.CharField(max_length=100)
    published_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title
```

- **Model Inheritance**:
  Every model class in Django must inherit from `models.Model`.
- **Fields**:
  - `CharField`: A string field for small-to-medium-sized strings.
  - `TextField`: A field for large text data.

- `DateTimeField`: Stores date and time.
- Other fields include `IntegerField`, `BooleanField`, `ForeignKey`, and many more.

- **Constraints**:
  - `max_length`: Specifies the maximum length for fields like `CharField`.
  - `auto_now_add`: Automatically sets the field to the current date/time when the object is first created.

- **The `__str__` method**:
  This is a string representation of the object (used in the Django admin interface and elsewhere).

Behind the scene

Once you've defined your models, Django generates SQL to create necessary database tables.

```
CREATE TABLE blog_post (
    id INT PRIMARY KEY AUTOINCREMENT,
    title VARCHAR(255),
    content TEXT,
    author VARCHAR(100),
    published_date DATETIME
);
```

▼ Migrations

▼ Introduction

Migrations in Django are a system that manages changes to the database schema over time. They allow to:

- Version control the database schema
- Safely apply changes to the database as Django models evolve.

Migrations are closely tied to the models defiled in the `models.py` files. Whenever a model is modified (e.g., add a field, rename a field, or change fields type), Django create a migration to apply these changes to the database.

▼ F.A.Q

- What problem migrations solve ?

  Schema evolution: As your application grows, you need to modify your database schema to accommodate new Features. Migrations handle it in a controlled way.

  Version control: You can track changes to your database schema and apply appropriate migrations on different environments (e.g., development, testing, production) in a consistent manner.

- Key migration commands ?

  `python3 manage.py makemigrations` create new migration files based on changes in the `models.py`.

`python3 manage.py migrate` applies migrations to the database, syncing models to the database schema.

- How do migrations work step-by-step ?
  - **Model Changes**: You modify or add fields to your models in the `models.py` file.
  - **Generate Migrations**: Run `makemigrations`. Django detects changes and creates migration files in the `migrations/` folder of each app.
  - **Migration Files**: Each migration file is a Python script describing the changes. Django generates raw SQL statements behind the scenes based on these files.
  - **Apply Migrations**: Run `migrate` to apply the changes to your database.
  - **Migration History**: Django tracks migrations applied to the database in a special table called `django_migrations`, ensuring migrations are applied only once.
- What happens if you delete migrations files (not recommended !) ?

  Database Schema: The database changes already applied via migrations will remain intact (nothing breaks in the database).

  Future Migrations: Without the migration history (the deleted files), Django may not be able to track future changes correctly. Deleting migration files might cause issues when attempting to apply new migrations.

  If you want to delete migrations and reset them, you need to carefully manage this by deleting the `django_migrations` table entries and recreating the migration files in sync with the current schema. This can be tricky and is generally discouraged unless absolutely necessary.
- What happens if you modify the Database directly (Without migrations) ?

  Database Changes: If you modify the database manually (e.g., using SQL in PostgreSQL), Django will be unaware of those changes.

  Desync Between Models and Database: The models and the database schema will fall out of sync. For example, if you rename a column in the database but don't reflect the change in `models.py`, Django will still expect the old column name, leading to errors.

  Migrations Won't Reflect Changes: Future migrations won't account for manual changes in the database, and you might face issues when running `migrate`.

  It is always recommended to make changes via migrations (or through Django's ORM) rather than directly in the database to avoid these problems.

▼ Common Django Field Types

- `CharField`: Fixed-length strings (e.g., name, email)
- `TextField`: Larger text (e.g., blog posts, comments)
- `IntegerField`: Stores integers
- `BooleanField`: True/False values
- `DateTimeField`: Date and time
- `EmailField`: Validates email format

- `ForeignKey`: For many-to-one relationships (e.g., one author, many posts)
- `ManyToManyField`: For many-to-many relationships (e.g., a post can have many tags)

▼ Model Managers

Django **Model Manages** are essential part of Django's ORM and are responsible for providing the database query interface for Django models. Every Django model has a default manager called `objects`, which provides methods like `all(), filter(), get()` , and other to interact with the database.

Definition

A model manager is a class that manages database queries related to a model. Model manages are customization, own manager can be created to extend or modify query behavior.

Custom managers are helpful when:

- You have complex or repetitive queries.
- You want to abstract business logic (like filtering published posts) in a reusable way.
- You need to add specialized query methods that your views or other parts of the application will use frequently.

▼ Creating a custom manager

To create a new custom manager, you define a new class that inherits from `models.Manager`. This allows you to add your own methods that extend or customize the default behavior.

Let's create a simple model manager.

```
from django.db import models

class PublishedPostManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(is_published=True)

class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.CharField(max_length=100)
    is_published = models.BooleanField(default=False)
    published_date = models.DateTimeField(null=True, blank=True)

    # Link the custom manager
    published = PublishedPostManager()

    def __str__(self):
        return self.title
```

- **Custom Manager (`PublishedPostManager`):**
  This custom manager overrides the `get_queryset()` method to return only posts where `is_published=True`.

- **Custom Manager Assignment**:
  Instead of using
  `Post.objects`, you would use `Post.published` to only retrieve published posts.

▼ Using a custom manager

After linking the custom manager class to the model, we can access it as a model's attribute which inherits methods for working with the manager.

```
# Using the default manager (will return all posts)
all_posts = Post.objects.all()

# Using the custom manager (will return only published posts)
published_posts = Post.published.all()
```

▼ Multiple mangers on a model

You can have multiple managers on the same model. For example, you might want a default manager that returns all posts, and a custom manager that returns only published posts.

```
class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.CharField(max_length=100)
    is_published = models.BooleanField(default=False)
    published_date = models.DateTimeField(null=True, blank=True)
    # Default manager (Post.objects)
    objects = models.Manager()
    # Custom manager for published posts
    published = PublishedPostManager()
    def __str__(self):
        return self.title
```

Here, `Post.objects` will return all posts (published and unpublished), while `Post.published` will return only published posts.

▼ Adding custom query methods in a manager

Custom managers can also include additional methods that return filtered query sets or perform actions. For example, lets add a method that returns all posts written by a particular author:

```
class PostManager(models.Manager):
    def by_author(self, author_name):
        return self.get_queryset().filter(author=author_name)

class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.CharField(max_length=100)
```

```
    is_published = models.BooleanField(default=False)
    published_date = models.DateTimeField(null=True, blank=True)

    # Custom manager
    objects = PostManager()

    def __str__(self):
        return self.title
```

Now, you can query posts my author using the custom method:

```
# Get all posts by the author "John Doe"
johns_posts = Post.objects.by_author("John Doe")
```

▼ Using Model Managers in the Admin

You can also use custom managers in Django's Admin interface. For example, you need admin interface to display only published posts.

```
class PostAdmin(admin.ModelAdmin):
    def get_queryset(self, request):
        qs = super().get_queryset(request)
        return qs.filter(is_published=True)

admin.site.register(Post, PostAdmin)
```

▼ The QuerySet class

Django Mangers return a `QuerySet` object, and the `get_queryset()` method is a key part of that. A QuerySet is a collections of database queries that Django will execute. By customizing the `get_queryset()` method in a manager, you control what data the manger returns.

If you need more complex logic, you can create custom QuerySets.

▼ Custom QuerySet

```
class PostQuerySet(models.QuerySet):
    def published(self):
        return self.filter(is_published=True)

class PostManager(models.Manager):
    def get_queryset(self):
        return PostQuerySet(self.model, using=self._db)

class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.CharField(max_length=100)
    is_published = models.BooleanField(default=False)

    objects = PostManager()
```

```
    def __str__(self):
        return self.title
```

Allows to chain queries in a flexible manner

```
published_posts = Post.objects.published()
```

▼ Query optimization

Key Techniques for Optimizing Django Queries:

1. **Lazy Querysets**
2. **Using `select_related` and `prefetch_related`**
3. **Avoiding the N+1 Problem**
4. **Using Query Aggregation**
5. **Database Indexing**
6. **Limiting the Fields Retrieved**
7. **Bulk Operations**
8. **Defer and Only for Large Models**

▼ Lazy Querying

In Django, querysets are lazy. This means that no database query is actually executed until the queryset is evaluated. This feature prevents unnecessary queries and allows you to chain queryset methods without hitting the database repeatedly.

```
# No database hit yet
posts = Post.objects.all()
# Database query executed when queryset is evaluated (e.g., in a loop)
for post in posts:
    print(post.title)
```

This laziness can be useful, but it's important to be mindful of when queries are evaluated (e.g., in templates or when passing a queryset to a function).

▼ Using `select_related` and `prefetch_related`.

One of the most common performance problem in Django is the N+1 problem. This occurs when your code queries related modes in a loop, resulting in multiple database hits instead of fetching all the related data in a single query.

`select_related` is used for foreign key and one-to-one relationships. It performs an SQL join and includes the related object in the same query, reducing the number of db hits.

```
class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
```

```
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
# Without select_related (N+1 problem):
posts = Post.objects.all()
for post in posts:
    print(post.author.name)  # This will execute one query per post (N
# With select_related (solves N+1 problem):
posts = Post.objects.select_related('author')
for post in posts:
    print(post.author.name)  # This will execute only one query
```

`prefetch_related` is used for M:M, and reversed foreign key relationships. It performs separated queries for the related objects and does the joining in Python rather than in the database.

```
class Post(models.Model):
    title = models.CharField(max_length=255)
    tags = models.ManyToManyField('Tag')
# Without prefetch_related:
posts = Post.objects.all()
for post in posts:
    print(post.tags.all())  # Multiple queries for each post (N+1 prob
# With prefetch_related:
posts = Post.objects.prefetch_related('tags')
for post in posts:
    print(post.tags.all())  # Executes just two queries (one for posts
```

▼ Avoiding N+1 problem

The N+1 problem happens when the code ends up make one query to fetch the main object (1 query) and additional queries for each related object (N queries). This typically occurs in a loop over related objects without using `select_related` and `prefetch_realted`.

▼ Using Query Aggregation

Django provides powerful aggregation functions to compute values like count, sum, average, etc.

```
from django.db.models import Count, Avg

# Get the total number of posts
total_posts = Post.objects.count()
# Get the average number of posts per author
average_posts_per_author = Post.objects.values('author').annotate(tota
# Total number of posts for each author
posts_per_author = Post.objects.values('author__name').annotate(post_c
```

By using aggregation, you're leveraging the database to compute totals or averages, which is more efficient than fetching all the data into Python and performing the computation manually.

▼ Database indexing

Indexes are crucial database optimization technique that makes lookups and joins falter. Django allows to specify which fields should be indexed at the model level.

To add an index to a field, use the `index=True` option in the model's field definition:

```
class Post(models.Model):
    title = models.CharField(max_length=255, db_index=True)  # Adds ar
    content = models.TextField()
```

Compound, and unique indexes are supported as well by using the Meta class.

```
class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    class Meta:
        indexes = [
            models.Index(fields=['title', 'author']),
        ]
```

Adding the right indexes can significantly improve query performance, but keep in mind that indexes slow down inserts and updates. Only index fields that are frequently queried.

▼ Limiting the fields retrieved.

Sometimes all database fields are not needed, especially if a table has a lot of columns or large data. You can optimize performance by limiting the fields retrieved from the database using the `only()` and `defer()` methods.

`only()` This methods retrieves only the fields specified, reducing the amount of data fetched.

```
# Fetch only the 'title' and 'published_date' fields
posts = Post.objects.only('title', 'published_date')
```

`defer()` This method retrieves all fields except the one specified. It's useful when you want most of the data but want to avoid fetching large fields file `TextField`.

```
# Fetch all fields except 'content'
posts = Post.objects.defer('content')
```

These methods can reduce memory usage and query execution time by retrieving only the necessary data.

▼ Bulk operations

When inserting or updating multiple records, Django's default behavior is to issue one query per record. This can lead to inefficiency in large databases. You can use b**ulk operation** like `bulk_create()` or `bulk_update()` to insert or update multiple records within a single query.

`bulk_create()`

```
# Insert multiple posts at once
Post.objects.bulk_create([
    Post(title='Post 1', content='Content 1', author=author),
    Post(title='Post 2', content='Content 2', author=author),
])
```

`bulk_update()`

```
# Update multiple posts at once
posts = Post.objects.filter(author=author)
for post in posts:
    post.content = 'Updated content'
Post.objects.bulk_update(posts, ['content'])
```

These operations are much more efficient than looping over each object and calling `save()`.

▼ Example of optimizing

Imagine you have a blog with many posts, each of which has an author and tags. You want to display the titles, authors, and tags of the most recent 10 posts.

```
# Optimized query
posts = (
    Post.objects.select_related('author')   # Avoids N+1 for the 'auth
        .prefetch_related('tags')           # Prefetches tags to avoi
        .only('title', 'author__name')      # Fetches only the necessa
        .order_by('-published_date')        # Orders by the most rece
        .all()[:10]                         # Limits the result to 10
)
```

This query optimizes:

- Foreign key retrieval with `select_related()`.
- Many-to-many retrieval with `prefetch_related()`.
- Only fetching necessary fields with `only()`.
- Limiting the number of posts with `[:10]`.

▼ Reversed foreign key

Understanding the **foreign key**. A **foreign key** is used to define a one-to-many relationship between two models. The model that holds the foreign key field is called a “child” or “ related” model, and the other is the “parent” or “referenced” model. A foreign key allows a “child” models to access the “parent” model fields.

The **reversed foreign key** allows the “referenced” model to access the “related” model’s fields.

▼ Example to illustrate reversed foreign key

Let’s consider a simple example where we have two models: `Author` and `Post`. Each post belongs to a single author (one-to-many relationship), but an author can have many posts.

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.ForeignKey(Author, on_delete=models.CASCADE)

    def __str__(self):
        return self.title
```

- Each `Post` has a foreign key field (`author`), which links it to an `Author`.
- Each `Author` can have multiple `Post` objects, but a `Post` can only have one `Author`.

▼ Forward Foreign Key Access

From the `Post` model, you can easily access the related `Author` using the foreign key relationship.

```
post = Post.objects.get(id=1)
print(post.author.name)  # Access the author of the post
```

▼ Reverse Foreign Key access

The **reverse foreign key** is how the **Author** model accesses all related `Post` objects. This works because Django automatically creates a **reverse relation** for each foreign key.

By default, Django creates this reverse relation as `related_name` if you don't explicitly set it. In our case, the default reverse relation for `Post` objects on the `Author` model would be `author.post_set` (model name in lowercase with `_set` suffix).

```
author = Author.objects.get(id=1)
posts = author.post_set.all()  # Get all posts written by this author
for post in posts:
    print(post.title)
```

- `author.post_set.all()` retrieves all `Post` objects where the `author` field matches this specific `Author`.
- This is the reverse access from `Author` to `Post`.

▼ Using `related_name` to customize the Reversed Foreign Key

By default, Django uses the pattern `<model_name>_set` to refer to the reverse foreign key. However, you can customize this by using the `related_name` attribute when defining the `ForeignKey`.

```
class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
    author = models.ForeignKey(Author, on_delete=models.CASCADE, relat

    def __str__(self):
        return self.title
```

In this case, you've set the `related_name` to `'posts'`. Now, instead of using `author.post_set`, you can access the related posts like this:

```
author = Author.objects.get(id=1)
posts = author.posts.all()  # Access all posts related to this author
for post in posts:
    print(post.title)
```

▼ In templates

In Django you can access objects by Reversed Foreign Key even from the templates.

```
<h1>{{ author.name }}</h1>

<ul>
    {% for post in author.posts.all %}
        <li>{{ post.title }}</li>
    {% endfor %}
</ul>
```

▼ Querying

You can also perform queries using reverse foreign keys. For example, you might want to get all authors who have written more than 5 posts:

```
from django.db.models import Count
authors = Author.objects.annotate(post_count=Count('posts')).filter(p
```

Here, `Count('posts')` counts the number of posts related to each author, and then the queryset filters to only authors with more than 5 posts.

▼ Views

▼ Generic views (CBVs)

Built in Class bases views. Their description and use cases. Generic views follow the DRY (Don't Repeat Yourself) principle by encapsulating common patterns, allowing developers to avoid writing repetitive code.

1. ListView

- **Description**: Renders a list of objects from a given model or queryset.

- **Use Case**: Use `ListView` when you need to display a list of items, such as a list of blog posts, products, or users.
- **Example**: Displaying a paginated list of articles in a blog application.

2. DetailView

- **Description**: Renders a detail page for a single object, typically identified by a primary key or slug.
- **Use Case**: Use `DetailView` when you need to show detailed information about a single object, like a single blog post or a user profile.
- **Example**: Viewing the details of a specific product in an e-commerce site.

3. CreateView

- **Description**: Provides a form for creating a new object and saves the object upon successful submission.
- **Use Case**: Use `CreateView` to simplify the process of adding new objects to the database.
- **Example**: Creating a new post in a blogging application.

4. UpdateView

- **Description**: Provides a form for editing an existing object and updates the object upon successful submission.
- **Use Case**: Use `UpdateView` to allow users to edit existing objects, like editing a user profile or updating a product's details.
- **Example**: Editing an existing comment on a blog post.

5. DeleteView

- **Description**: Provides a confirmation page for deleting an object and deletes the object upon confirmation.
- **Use Case**: Use `DeleteView` when you need to provide functionality to delete objects from the database.
- **Example**: Deleting a user account or a specific blog post.

6. TemplateView

- **Description**: Renders a template without requiring a model. It’s used for displaying static content.
- **Use Case**: Use `TemplateView` when you need to render a static HTML page that doesn’t require interaction with a database model.
- **Example**: Displaying an "About Us" page or a "Terms and Conditions" page.

7. RedirectView

- **Description**: Redirects to a given URL or named URL pattern.
- **Use Case**: Use `RedirectView` to perform URL redirects, such as redirecting after a form submission or handling old URLs.

- **Example**: Redirecting from an old URL to a new URL after a page has been moved.

8. FormView

- **Description**: Renders a form and processes form submissions, without associating it directly with a model.
- **Use Case**: Use `FormView` when you need to handle forms not directly tied to a database model, like contact forms or search forms.
- **Example**: Implementing a contact form on a website.

9. ArchiveIndexView

- **Description**: Displays a list of objects grouped by date (usually year or month).
- **Use Case**: Use `ArchiveIndexView` for date-based archives, like blog archives grouped by year.
- **Example**: A blog page displaying all posts from the current year.

10. YearArchiveView

- **Description**: Displays objects from a specific year.
- **Use Case**: Use `YearArchiveView` to create archives grouped by year.
- **Example**: Showing all blog posts from a particular year.

11. MonthArchiveView

- **Description**: Displays objects from a specific month and year.
- **Use Case**: Use `MonthArchiveView` to create monthly archives of objects.
- **Example**: Showing all articles published in March 2024.

12. WeekArchiveView

- **Description**: Displays objects from a specific week.
- **Use Case**: Use `WeekArchiveView` to group objects by week, which is helpful for creating weekly summaries or round-ups.
- **Example**: Showing all posts from the first week of April 2024.

13. DayArchiveView

- **Description**: Displays objects from a specific day.
- **Use Case**: Use `DayArchiveView` for daily archives, like a day-by-day archive of blog posts.
- **Example**: Displaying posts from April 1, 2024.

14. TodayArchiveView

- **Description**: Displays objects from the current day.
- **Use Case**: Use `TodayArchiveView` to show items created today, which is useful for daily updates.
- **Example**: Showing all articles published today.

15. DateDetailView

- **Description**: Displays a single object based on a date and a slug.
- **Use Case**: Use `DateDetailView` to display a specific object identified by a date.
- **Example**: Displaying a blog post from a specific day.

▼ Function based views

▼ Templates

▼ Forms

▼ Handling data in a form

▼ Clean data

`cleaned_data` is a dictionary-like attribute that contains the validated data from the form's fields. After a form has been submitted and its data has been validated (using the `is_valid()` method), the `cleaned_data` dictionary is populated with the cleaned (validated and converted) data from the form.

▼ Custom data cleaners

```
class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)

    def clean_email(self):
        email = self.cleaned_data.get('email')
        if not email.endswith('@example.com'):
            raise forms.ValidationError('Email must be from example.c
        return email
```

```
class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)

    def clean(self):
        cleaned_data = super().clean()
        name = cleaned_data.get('name')
        message = cleaned_data.get('message')

        if name and message:
            if 'urgent' in message and name != 'Admin':
                raise forms.ValidationError('Only Admin can send urge
```

```
        return cleaned_data
```

▼ is_valid

- **Field Validation:**
  - When you call `form.is_valid()`, Django goes through each field in the form and runs the validation logic for that field. This includes built-in validations like checking if a required field is filled in, or if an email is correctly formatted, as well as any custom validation logic you may have added.
  - If a field's data passes validation, it is added to the `cleaned_data` dictionary.
- **Data Cleaning and Conversion:**
  - During validation, the data is also "cleaned" and converted into the appropriate Python data types. For example, if a form field is a `DateField`, the data entered as a string (e.g., "2024-09-04") is converted into a Python `datetime.date` object and stored in `cleaned_data`.
- **Accessing `cleaned_data`:**
  - Once the form is validated, you can access the `cleaned_data` dictionary to retrieve the validated data and perform further processing, such as saving it to a database or using it in some other way.

▼ Errors

1. ValidationError

- **What It Is:**
  - `ValidationError` is the most common error raised during form validation. It occurs when the data provided by the user does not meet the validation criteria set by the form fields or custom validation methods.
- **Examples:**
  - Entering an invalid email address in an `EmailField`.
  - Providing a string where an integer is expected in an `IntegerField`.

2. TypeError

- **What It Is:**
  - `TypeError` can occur if you attempt to perform an operation on a data type that isn't compatible with that operation, often resulting from incorrect data types being passed to form fields.
- **Examples:**
  - Passing a list instead of a string to a `CharField`.
  - Using an integer where a `DateField` expects a date object.
- **Common Scenarios:**
  - When writing custom validation logic and accidentally performing operations on the wrong data type.

3. ValueError

- **What It Is:**
  - `ValueError` occurs when a function receives an argument of the right type but with an inappropriate value, particularly in custom validation logic.
- **Examples:**
  - Trying to convert a string that doesn't represent a valid number to an integer.
- **Common Scenarios:**
  - When manually processing form data in a view or within custom `clean()` methods.

4. KeyError

- **What It Is:**
  - `KeyError` can occur when you try to access a key in a dictionary that doesn't exist. In form validation, this might happen if you attempt to access a field in `cleaned_data` that hasn't been validated yet or doesn't exist in the form.
- **Examples:**
  - Accessing a non-existent key in `cleaned_data`.
- **Common Scenarios:**
  - In custom `clean()` methods where you access multiple fields together without checking if they exist first.

5. AttributeError

- **What It Is:**
  - `AttributeError` occurs when you try to access or call an attribute or method that doesn't exist on an object. In forms, this can happen if you assume a certain field or method exists when it does not.
- **Examples:**
  - Attempting to call a method that is not defined on a form field.
- **Common Scenarios:**
  - When you have a typo in field names or when the form or model object is incorrectly referenced.

6. IntegrityError

- **What It Is:**
  - `IntegrityError` occurs when a form tries to save data that violates database constraints, such as uniqueness or foreign key constraints.
- **Examples:**
  - Trying to create a new user with a username that already exists if the `username` field is marked as unique.
- **Common Scenarios:**

- When saving form data directly to the database, especially in `ModelForm`.

7. MultiValueDictKeyError

- **What It Is:**
  - `MultiValueDictKeyError` is a subclass of `KeyError` and occurs when you try to access a key in a Django request's `POST` or `GET` data (which are `MultiValueDicts`) that doesn't exist.
- **Examples:**
  - Accessing a non-existent key in `request.POST` or `request.GET`.
- **Common Scenarios:**
  - When processing form data directly from the request object without using `request.POST.get()` or `request.GET.get()` methods that handle missing keys more gracefully.

8. FieldError

- **What It Is:**
  - `FieldError` occurs when there is a problem with a form field, such as when trying to set a form field that doesn't exist or when a field is improperly configured.
- **Examples:**
  - Trying to set a field that is not part of the form.
- **Common Scenarios:**
  - When dynamically adding or modifying form fields.

9. ImproperlyConfigured

- **What It Is:**
  - `ImproperlyConfigured` occurs when Django is not properly configured, which could include issues in form setup or field configuration.
- **Examples:**
  - Misconfiguring a form field that expects a certain setting to be present.
- **Common Scenarios:**
  - When configuring forms or integrating them with other Django components.

▼ Model Form

A **ModelForm** in Django is a class that automatically generates a form based on a Django model. It simplifies the process of creating forms that interact with the database, allowing developers to create forms for model data without manually defining the form fields.

Use Cases of ModelForm

1. **CRUD Operations**: When you need to create, update, or delete records in the database through a form, a ModelForm can be used to automatically generate the form fields that correspond to the model fields.

2. **Form Validation**: ModelForms include built-in validation based on the model's field constraints (e.g., `max_length`, `unique`, `blank`, etc.), reducing the amount of custom validation code you need to write.
3. **Consistency**: Ensures consistency between the form and the model by reusing the model's field definitions and validation logic.
4. **Admin Interface**: Django's admin interface uses ModelForms to create forms for models automatically.

How to Use a ModelForm

1. **Define a Model**: First, you need to have a Django model that represents the data structure:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
    isbn = models.CharField(max_length=13, unique=True)

    def __str__(self):
        return self.title
```

2. **Create a ModelForm**: You then create a ModelForm class that specifies the model it corresponds to:

```
from django import forms
from .models import Book

class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ['title', 'author', 'published_date', 'isbn']
```

   - `model`: Specifies which model the form is based on.
   - `fields`: Specifies which model fields should be included in the form. You can also exclude fields using `exclude = ['field_name']`.

3. **Use the ModelForm in Views**: Finally, you use this ModelForm in your views to handle form submissions, validation, and saving the data to the database:

```
from django.shortcuts import render, redirect
from .forms import BookForm

def add_book(request):
    if request.method == 'POST':
        form = BookForm(request.POST)
```

```
        if form.is_valid():
            form.save()  # Save the new book to the database
            return redirect('book_list')  # Redirect to a list of boo
    else:
        form = BookForm()
    return render(request, 'add_book.html', {'form': form})
```

```
<!-- templates/add_book.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Add Book</title>
</head>
<body>
    <h2>Add a New Book</h2>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Save</button>
    </form>
</body>
</html>
```

Advanced Use Cases

- **Overriding Fields**: You can override specific fields in the ModelForm to customize their behavior.
- **Custom Validation**: You can add custom validation methods for specific fields or the entire form.
- **Partial Forms**: You can use `fields` or `exclude` to create forms that only include a subset of the model's fields.

```
class BookForm(forms.ModelForm):
    title = forms.CharField(widget=forms.TextInput(attrs={'placeholde

    class Meta:
        model = Book
        fields = ['title', 'author', 'published_date', 'isbn']
```

▼ Widgets

Widgets in Django Forms are the building blocks that represent HTML form elements such as `<input>`, `<select>`, and `<textarea>`. They are responsible for rendering these form fields on the frontend and handling the user input. Each form field in Django is associated with a widget that determines how it will be displayed in the HTML and how the input data will be processed.

▼ Example using widgets

```
from django import forms

class UserRegistrationForm(forms.Form):
    username = forms.CharField(max_length=100, widget=forms.TextInput(
    password = forms.CharField(widget=forms.PasswordInput(attrs={'clas
    email = forms.EmailField(widget=forms.EmailInput(attrs={'class':
    birth_date = forms.DateField(widget=forms.SelectDateWidget(years=r
```

In this example:

- `widget=forms.TextInput(...)`: Specifies a `TextInput` widget for the `username` field, with additional HTML attributes like `class` and `placeholder`.
- `widget=forms.PasswordInput(...)`: Uses a `PasswordInput` widget for the `password` field, rendering it as a password input box.
- `widget=forms.EmailInput(...)`: Customizes the `EmailField` with an `EmailInput` widget.
- `widget=forms.SelectDateWidget(...)`: Provides a dropdown for selecting a date with a custom range of years.

▼ Built-it widgets

1. `TextInput`: `widget=forms.TextInput(attrs={'class': 'form-control'})`
2. `PasswordInput`: `widget=forms.PasswordInput(attrs={'class': 'form-control'})`.
3. `EmailInput`: `widget=forms.EmailInput(attrs={'placeholder': 'Enter your email'})`.
4. `TextArea`: `widget=forms.Textarea(attrs={'rows': 4, 'cols': 15})`.
5. `CheckboxInput`: `widget=forms.CheckboxInput()`.
6. `Select`: `widget=forms.Select(choices=[('1', 'One'), ('2', 'Two')])`.
7. `RaidioSelect`: `widget=forms.RadioSelect(choices=[('M', 'Male'), ('F', 'Female')])`.
8. `FileInput`: `widget=forms.FileInput()`.
9. `DateInput`: `widget=forms.DateInput(attrs={'type': 'date'})`
10. `SelectDatewidget`: `widget=forms.SelectDateWidget(years=range(1980, 2025))`.
11. `TimeInput`: `widget=forms.TimeInput(attrs={'type': 'time'})`.
12. `NumberInput`: `widget=forms.NumberInput()`.

▼ Customizing widgets

You can customize widgets by passing additional attributes using the `attrs` argument. This allows you to add CSS classes, placeholder text, or any other HTML attributes to the rendered widget.

```
class CommentForm(forms.Form):
    comment = forms.CharField(widget=forms.Textarea(attrs={
        'class': 'form-control',
        'rows': 5,
```

```
        'placeholder': 'Enter your comment here...'
    }))
```

Build custom widgets

If the built-in widgets don't meet your requirements, you can create custom widgets by subclassing `django.forms.widgets.Widget` and implementing the necessary methods.

```
from django.forms.widgets import Widget

class CustomTextInput(Widget):
    template_name = 'custom_widgets/custom_text_input.html'

    def __init__(self, attrs=None):
        super().__init__(attrs)

    def get_context(self, name, value, attrs):
        context = super().get_context(name, value, attrs)
        context['custom_attribute'] = 'custom_value'
        return context
```

▼ Validation and custom validators

Validation levels:

▼ Form

Form Validators: These are applied to the entire form. They are typically used when you need to validate the relationship between multiple fields or the overall consistency of the form data.

Form validators are typically used when the validation logic depends on more than one field. You can override the `clean` method in your form to implement form-level validation.

```
from django import forms
from django.core.exceptions import ValidationError

class SignupForm(forms.Form):
    username = forms.CharField(max_length=100)
    email = forms.EmailField()
    confirm_email = forms.EmailField()

    def clean(self):
        cleaned_data = super().clean()
        email = cleaned_data.get("email")
        confirm_email = cleaned_data.get("confirm_email")

        if email != confirm_email:
            raise ValidationError("Emails do not match")
```

In this example, the form checks if the `email` and `confirm_email` fields match. If they don't, a `ValidationError` is raised, which will prevent the form from being submitted and display an error message to the user.

▼ Using Validators in Model Forms

When using Model Forms, validators can be applied directly to the model fields in the model definition, and they will automatically be applied to the corresponding form fields.

```
from django.db import models
from django.core.validators import MinLengthValidator

class User(models.Model):
    username = models.CharField(max_length=100, validators=[MinLength\
    email = models.EmailField()

class UserForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ['username', 'email']
```

In this case, the `MinLengthValidator` is applied to the `username` field at the model level, and it will be enforced when the field is used in a form.

▼ Field

Field Validators: These are applied to individual form fields. They are used to ensure that a single field meets specific criteria, such as being a valid email address, having a minimum length, or matching a regular expression pattern.

Django provides a set of built-in validators that you can attach to form fields. Additionally, you can create custom validators if the built-in ones don't meet your needs.

Built-in Field Validators

Some common built-in validators include:

- `MinLengthValidator`: Ensures the input has at least a minimum number of characters.
- `MaxLengthValidator`: Ensures the input does not exceed a maximum number of characters.
- `EmailValidator`: Ensures the input is a valid email address.
- `URLValidator`: Ensures the input is a valid URL.
- `RegexValidator`: Ensures the input matches a specified regular expression.

```
from django import forms
from django.core.validators import MinLengthValidator, EmailValidat

class UserRegistrationForm(forms.Form):
    username = forms.CharField(max_length=100, validators=[MinLengt
    email = forms.EmailField(validators=[EmailValidator()])
    password = forms.CharField(widget=forms.PasswordInput, validat
```

Custom validators

```
from django import forms
from django.core.exceptions import ValidationError

def validate_even(value):
    if value % 2 != 0:
        raise ValidationError(f'{value} is not an even number')

class NumberForm(forms.Form):
    number = forms.IntegerField(validators=[validate_even])
```

In this example, the `validate_even` function checks if the input value is an even number. If it's not, a `ValidationError` is raised, which will be displayed to the user.

Handling validation errors:

▼ Form-sets

A **Formset** in Django is a layer of abstraction to work with multiple forms on the same page. Instead of handling each form individually, formsets allow you to manage a collection of forms as a single entity, making it easier to create, update, and delete multiple instances of a model or process multiple forms simultaneously.

▼ Formset parameters

1. `extra: int | None = None` the number of forms displayed.
2. `max_num: int | None = None` : Limits the max number of forms can be displayed.
3. `can_order: bool = False` : Adds an order field for each form in a formset, allowing user to specify the order of the forms.
4. `can_delete: bool = True` : Adds a checkbox to each form that allows form to be marked for deletion when the formset is submitted.
5. `min_num: int = 0` : Enforces a minimum number of forms that must be submitted in the formset.
6. `validation_min: bool = False` : raises a validation error is `min_num` rule is violated submitting the form.
7. `validate_max: bool = False` : Raises a validation error if submitting the form the `max_num` rule was violated.
8. `formset: FormasetClass` : Allows to specify custom formset class...
9. `fields: list[str] | None = None` : Specifies which fields should be included in the form when using a `ModelFormSet`.
10. `exclude` : Specifies which fields should be excluded from the form when using a `ModelFormSet`.
11. `widgets: dict` :Allows you to override the default widgets for specific fields in the formset.

Types of Formsets

▼ `Formset` (basic)

These are used to manage multiple instances of a standard Django form (not tied to any model). You use the `formset_factory` method to create a basic formset.

```
from django import forms
from django.forms import formset_factory

class ContactForm(forms.Form):
    name = forms.CharField()
    email = forms.EmailField()

ContactFormSet = formset_factory(ContactForm, extra=3)
```

Here, `ContactFormSet` is a formset that will display three `ContactForm` instances by default.

Parameters:

▼ `ModelFormset`

These are used to manage multiple instances of a form tied to a Django model. The forms in a model formset correspond to model instances, and they can be used to create, update, or delete multiple model instances. You use the `modelformset_factory` method to create a model formset.

```
from django import forms
from django.forms import modelformset_factory
from myapp.models import Author

class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        fields = ['name', 'email']

AuthorFormSet = modelformset_factory(Author, form=AuthorForm, extra=2
```

This `AuthorFormSet` will display forms for two new `Author` instances by default, alongside forms for existing authors.

▼ Inline form-set (1:N)

These are a specialized version of model formsets used to manage related objects, typically for handling related models in a parent-child relationship (e.g., managing books related to a specific author). Inline formsets are created using the `inlineformset_factory` method.

Inline formsets are useful for managing related objects. For example, if you have an `Author` model and a related `Book` model, you can manage books related to a specific author.

```
from django.forms import inlineformset_factory
from myapp.models import Author, Book
```

```
BookFormSet = inlineformset_factory(Author, Book, fields=('title', 'pu
```

This `BookFormSet` can be used to display and manage books related to a specific author.

Key Features

- `extra` **Parameter**: Controls how many empty forms are displayed in addition to the existing forms.
- `can_delete` **Parameter**: Allows you to include a checkbox in each form to mark it for deletion.
- `can_order` **Parameter**: Adds an ordering field to each form in the formset, useful for ordering objects in a list.

Use Cases

- **Basic Formset**: Useful for handling multiple forms that don't directly correspond to models, such as a survey with multiple questions.
- **Model Formset**: Ideal for batch-creating or updating model instances.
- **Inline Formset**: Perfect for managing related objects in a parent-child relationship, like handling multiple `Book` instances related to an `Author`.

Formset in a view

```
from django.shortcuts import render, redirect
from myapp.forms import ContactFormSet

def manage_contacts(request):
    if request.method == 'POST':
        formset = ContactFormSet(request.POST)
        if formset.is_valid():
            # Process the form data
            for form in formset:
                print(form.cleaned_data)
            return redirect('success')
    else:
        formset = ContactFormSet()
    return render(request, 'manage_contacts.html', {'formset': formset})
```

▼ File uploads

1. **Create a Form with a FileField**

   Django's `forms.FileField` is used to handle file uploads in forms. If you're also uploading images, you can use `forms.ImageField`.

   ```
   class UploadImageForm(forms.Form):
       title = forms.CharField(max_length=50)
       image = forms.ImageField()
   ```

2. Modify HTML form:

   Make sure that your HTML form includes the `enctype="multipart/form-data"` attribute, which is necessary for file uploads.

```
<form method="post" enctype="multipart/form-data">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Upload</button>
</form>
```

3. Handle the upload file in the views;

   In your view, you'll need to handle the uploaded file. You can access the uploaded file using `request.FILES`. Here's how to handle the file upload:

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from .forms import UploadFileForm

# Function to handle file upload
def handle_uploaded_file(f):
    with open(f'media/{f.name}', 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/')  # Redirect afte
    else:
        form = UploadFileForm()
    return render(request, 'upload_file.html', {'form': form})
```

4. Configure media settings:

```
STATIC_URL = 'static/'
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')

STATICFILES_DIRS = [
    BASE_DIR / 'static'
]
```

5. Configure Media URLs for production and debug:

```
from django.conf.urls.static import static
from django.urls import path, include, re_path
from django.conf import settings
import debug_toolbar
# NOTE: Add in production!
from django.conf.urls.static import static
from django.views.static import serve


urlpatterns = [
    # NOTE: Uncomment in production!
    # re_path(r'^media/(?P<path>.*)$', serve, {'document_root': setti
    # re_path(r'^static/(?P<path>.*)$', serve, {'document_root': sett
    path('admin/', admin.site.urls),
    path('posters/', include(('posters_app.urls', 'posters_app'), nam
    path('user_account/', include(('user_account_app.urls', 'user_acc
    path('user_auth/', include("django.contrib.auth.urls")),
]

# NOTE: Comment in Production!
if settings.DEBUG:
    urlpatterns += [path('__debug__/', include(debug_toolbar.urls))]
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.
```

6. Handle upload to the models.

▼ Security

Use CSRF token.

▼ Testing forms

Testing Django forms in `tests.py` involves verifying the behavior of the form's validation, including its success and failure conditions.

▼ Import

`from django.test import TestCase`

`from .forms import ContactForm`

▼ Test Valid Form Data

```
class ContactFormTest(TestCase):

    def test_contact_form_valid(self):
        form_data = {
            'name': 'John Doe',
            'email': 'john@example.com',
            'message': 'This is a test message.'
        }
```

```
        form = ContactForm(data=form_data)
        self.assertTrue(form.is_valid())
```

In this test:

- You pass valid data to the form and check if `form.is_valid()` returns `True`.

▼ Test Invalid Form Data

Here, you test the behavior of the form when given invalid or missing data. You can check that the form returns errors for the required fields or for any custom validation rules.

```
class ContactFormTest(TestCase):

    def test_contact_form_invalid(self):
        # Missing required fields
        form_data = {
            'name': '',
            'email': 'invalid-email',
            'message': ''
        }
        form = ContactForm(data=form_data)
        self.assertFalse(form.is_valid())
        self.assertEqual(len(form.errors), 3)  # We expect 3 errors: r
        self.assertIn('name', form.errors)
        self.assertIn('email', form.errors)
        self.assertIn('message', form.errors)
```

▼ Test Custom Validation

If your form contains custom validation methods (e.g., `clean_<field>()` methods), you should test those as well. Example: Suppose you have a custom validation method that checks if the email domain is `example.com`.

```
class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)

    def clean_email(self):
        email = self.cleaned_data.get('email')
        if not email.endswith('@example.com'):
            raise forms.ValidationError("Email must be from the domai
        return email
```

```
class ContactFormTest(TestCase):

    def test_contact_form_custom_email_validation(self):
        form_data = {
            'name': 'John Doe',
```

```
            'email': 'john@gmail.com',
            'message': 'This is a test message.'
        }
        form = ContactForm(data=form_data)
        self.assertFalse(form.is_valid())
        self.assertIn('email', form.errors)
        self.assertEqual(form.errors['email'], ["Email must be from t
```

▼ Test Bound Data

You can also test how the form binds data and initializes with it, ensuring that form fields hold the correct values when rendered back to the user after a failed submission.

```
class ContactFormTest(TestCase):

    def test_contact_form_bound_data(self):
        form_data = {
            'name': 'John Doe',
            'email': '',
            'message': 'This is a test message.'
        }
        form = ContactForm(data=form_data)
        self.assertFalse(form.is_valid())
        self.assertEqual(form.cleaned_data['name'], 'John Doe')
        self.assertEqual(form.cleaned_data['message'], 'This is a tes
        self.assertNotIn('email', form.cleaned_data)  # Email is inva
```

▼ Test Form Rendering (optional)

While not common, you can test how a form is rendered by checking if the expected fields are in the HTML output. This ensures that forms are displayed correctly.

```
class ContactFormTest(TestCase):

    def test_contact_form_rendering(self):
        form = ContactForm()
        rendered_form = form.as_p()
        self.assertIn('name', rendered_form)
        self.assertIn('email', rendered_form)
        self.assertIn('message', rendered_form)
```

▼ AJAX forms

Full Workflow:

- The user fills out the form and submits it.
- The form submission triggers an AJAX request.
- Django receives the AJAX request, validates the form, and returns a JSON response.
- If the form is valid, a success message is shown.

- If there are errors, they are displayed without reloading the page.

1. **Create Django Form:**

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)
```

2. **Create a view to handle the form:**

   The view will process the form submission and return a JSON response to the AJAX request. This view can handle both regular and AJAX form submissions.

```
from django.http import JsonResponse
from django.shortcuts import render
from .forms import ContactForm

def contact_view(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Process the form (e.g., send email, save to DB, etc.)
            # Return a JSON response if the form is valid
            return JsonResponse({'success': True})
        else:
            # Return form errors as JSON
            return JsonResponse({'success': False, 'errors': form.err
    else:
        form = ContactForm()

    return render(request, 'contact_form.html', {'form': form})
```

3. **Create URL pattern:**

```
from django.urls import path
from .views import contact_view

urlpatterns = [
    path('contact/', contact_view, name='contact'),
]
```

4. **Create HTML form:**

   In the template, create a form and add JavaScript to handle the AJAX submission.

```
<form id="contactForm" method="post" action="{% url 'contact' %}">
    {% csrf_token %}
```

```
    {{ form.as_p }}
    <button type="submit">Submit</button>
</form>

<!-- Display error or success message -->
<div id="formMessages"></div>

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script>
    $(document).ready(function() {
        $('#contactForm').on('submit', function(event) {
            event.preventDefault();  // Prevent the form from submitt

            // Serialize the form data
            var formData = $(this).serialize();

            // Send the form data via AJAX
            $.ajax({
                url: $(this).attr('action'),
                type: $(this).attr('method'),
                data: formData,
                dataType: 'json',
                success: function(response) {
                    if (response.success) {
                        // Display success message
                        $('#formMessages').html('<p>Form submitted su
                    } else {
                        // Display error messages
                        var errors = '';
                        for (var field in response.errors) {
                            errors += '<p>' + field + ': ' + response
                        }
                        $('#formMessages').html(errors);
                    }
                },
                error: function(xhr, errmsg, err) {
                    // If there's an error in the AJAX request
                    $('#formMessages').html('<p>There was an error su
                }
            });
        });
    });
</script>
```

5. **Handle CSRF Token:**

   In Django, CSRF protection is enabled by default. Make sure to include the CSRF token in the form as well as in the AJAX request.

If you're not using jQuery, you can include the CSRF token in the request header using the following JavaScript snippet:

```
function getCookie(name) {
    let cookieValue = null;
    if (document.cookie && document.cookie !== '') {
        const cookies = document.cookie.split(';');
        for (let i = 0; i < cookies.length; i++) {
            const cookie = cookies[i].trim();
            if (cookie.substring(0, name.length + 1) === (name + '=')
                cookieValue = decodeURIComponent(cookie.substring(nam
                break;
            }
        }
    }
    return cookieValue;
}

const csrftoken = getCookie('csrftoken');

$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!/^(GET|HEAD|OPTIONS|TRACE)$/.test(settings.type) && !thi
            xhr.setRequestHeader("X-CSRFToken", csrftoken);
        }
    }
});
```

6. **Return JSON response:**

   In the `contact_view`, return a `JsonResponse` for both success and error scenarios. The success response may include a success message, and the error response should include the form validation errors.

▼ Caching

Django provides a powerful Framework for caching. Django does not always need a caching backend like Redis, it is able to perform some cache operation on its own

Server side

▼ Caching backend supported by Django

1. **Local-memory Caching:**
   a. **Description:** This cache backend stores cached data in memory of the running Django server process. it is the simplest form of caching that does not require any dependencies.
   b. **Use Case:** low-traffic website or development environment where there are no concerns about cached data persistence after server restarts.
   c. **Example Configuration:**

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache'
    }
}
```

2. **File-Base Caching:**
    a. **Description:** Stores cache data in files on the local file system. This approach is straightforward and doesn't require additional software but can be slower than in-memory cache.
    b. **Use Case:** from low to medium traffic websites where the is more data that can fit in memory.
    c. **Example Configuration:**

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBased
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

3. **Database Caching:**
    a. **Description:** Uses the data base configured for Django application to store cache data. Ensure the data persistence after server reload. But slower than in-memory cache.
    b. **Use Cases:** When cache retrieval speed doesn't matter but data persistence across server restarts is necessary.
    c. **Example Configuration:**

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_cache_table',
    }
}
```

    Note: You must create the cache table in your database using the management command `python manage.py createcachetable`.

4. Dummy Caching:
    a. **Description:** The backend that doesn't actually cache anything.
    b. **Use Case:** Development environments when you want to disable caching without removing code.

c. **Example Configuration:**

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',
    }
}
```

▼ External Caching backends

Integrating an external caching backend like Redis or Memcached can significantly improve performance, especially for high-traffic sites.

Redis

1. Redis:
    a. **Description:** An in-memory database that can be user as a cache, message broker, and database.
    b. **Use Case:** ideal for high-traffic websites. Avail data persistence and replication.
    c. **Example Configuration:**

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
        'OPTIONS': {
            'CLIENT_CLASS': 'django_redis.client.DefaultClient',
        }
    }
}
```

2. **Memcached:**
    a. **Description:** A high-performance, distributed memory object caching system designed to speed up dynamic web applications by alleviating database load.
    b. **Use Case:** Great for applications where distributed caching is needed across multiple servers or processes.
    c. **Example Configuration:**

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.Memcached
        'LOCATION': '127.0.0.1:11211',
    }
}
```

Caching in the use:

▼ Cache objects in views

Client side

▼ Client side caching (general)

Clint side caching is done using the `Cache-control` heading.

How Client-Side Caching Works:

1. **Initial Request**: The first time a client requests a resource, the server sends it to the client, along with cache-control headers that specify how long the resource can be cached.
2. **Subsequent Requests**: On future visits, the browser checks if the cached resource is still valid (based on the cache-control headers). If it's valid, the browser uses the cached version instead of making a new request to the server.
3. **Expired Cache**: When a cache expires or the resource is updated, the browser fetches the updated resource from the server.

Cache-Control Headers:

- `Cache-Control`: Defines caching directives, like how long the resource should be cached (e.g., `max-age=3600` means cache for 1 hour).
- `ETag`: A unique identifier for a specific version of a resource. If the ETag changes, the browser knows to request a fresh version of the resource.
- `Expires`: Specifies an exact expiration time for the cached resource.
- `Last-Modified`: The last time the resource was changed. If the resource has been modified since the cached version, the browser fetches the new one.

▼ Client-Side Caching in Django:

Django itself does not directly control the browser's cache, but it provides tools to help manage client-side caching through **HTTP headers** and **static file management**:

1. **Setting Cache-Control Headers**: Django allows you to set cache-control headers using middleware or view decorators:
   - You can use the `@cache_control` decorator to set caching policies for specific views.

```
from django.views.decorators.cache import cache_control

@cache_control(max_age=3600, must_revalidate=True)
def my_view(request):
    # Your view logic here
```

2. **Static File Versioning**: When using Django's `django.contrib.staticfiles`, the `collectstatic` management command can add version identifiers (hashes) to static files to help with cache busting (forcing the browser to fetch new versions when files change).
   - By enabling the `STATICFILES_STORAGE` setting to use the `ManifestStaticFilesStorage`, you can automatically append a hash to static file names:

```
STATICFILES_STORAGE = 'django.contrib.staticfiles.storage.Manifest
```

3. **Middleware**:
    - Django provides the `UpdateCacheMiddleware` and `FetchFromCacheMiddleware` for server-side caching, but to manage client-side caching, you'd generally use a combination of headers (like `Cache-Control`) or rely on static file management.

▼ Authentication and Authorization

⚠ Django will not automatically disallow a User to get to an object the user has no permission to. You need to check if a User has a Permission to a certain object (like view object, etc.) manually.

Key features of Django's authentication and authorization system include:

- Built-in views and forms for user registration, login, and password management
- A user model that can be extended to include additional fields
- Decorators and mixins to restrict access to views based on permissions
- An admin interface for managing users, groups, and permissions

This system allows developers to implement complex access control schemes while keeping the codebase clean and maintainable.

Group management and user permissions in Django are powerful features that allow you to control access to different parts of your application. In order to grant a user or a group of users a permission a user needs to be registered and authenticated:

▼ Register & delete users

▼ Authentication

> https://docs.djangoproject.com/en/5.1/topics/auth/

Type:

▼ Session key based

You can authenticate users based on their session key. Each website visitor get a cookie 'session id' based on this cookie Django assign a `session_key` to each `session id'.

▼ Middleware

The session_key is not automatically granted to all Anonymous visitors but it can be enabled via Middleware that checks whether a visitor has a session_key or not, and if not the session key is generated and assigned before the request reached any of the views.

```
```

▼ Token based

▼ User based

▼ User registration (sign Up)

Django does not provide built-ins such as user Registration URL, so it is better to create an application for user, where everything related to users will be stored.

1. Create an application `user_account`. include in into `INSTALLED_APPS` in the `settings.py` and `urls.py`.
2. Create a view for user registration, so as template and add URL in the `urls.py`.
3. User creating view based on a built-in `UserCreatingForm` :

```
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import login

# Additionally you can login a newly create user.
def user_sign_up(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
                form.save() # Just create a user
            login(request, form.save()) # If want to login a use
            return redirect('user_account_app:view_user_account'
    else:
        form = UserCreationForm()

    return render(request, 'user_account_app/user_sign_up.html',
```

▼ Built-in

Django provides a build-in module for authentication.

Requirements:

1. URLs connection:
   `path('', include('django.contrib.auth.urls'))`
2. Create HTML pages at Projects's base templates folder in the `registration` sub folder:
   a. `login.html`
   b. `logout.html`
   c. `reg.html` or `sign-up.html` PS: Render this view a custom view, as Django does not support registration page on its own. Use `UserCreatingForm(request.POST)` for registering new users.

   Django will render these specific templates automatically whenever user needs to login or logout. Inside of the `login.html` place a POST form that renders `{{ form }}` .
3. Additional settings at `settings.py`
   `LOGIN_REDIRECT_URL = "/polls/"`

   `LOGOUT_REDIRECT_URL = "/registration"`

Django has built-in methods for managing User Auth.
`from django.contrib.auth import login, logout, authenticate`

1. `logint(request, user)`.Logs a user in.

Built-in methods

In order to redesign built-in pages you need to create a HTML page with the same name as the URI you want to redesign. E.g. `/accounts/password_reset [name='somethung']` =⇒ `password_reset.html`.

Django has many built-in methods like:

1. `password_reset`
2. `password_change` etc.

▼ Custom

You can write your own login logic using `AuthenticationForm` . Do not forget to create the `login_template.html` and add the login view to the `urls.py`.

```
from django.contrib.auth.forms import AuthenticationForm
from django.contrib.auth import login

def login(request):
    if request.method == "POST":
        form = AuthenticationForm(data=request.POST)
        if form.is_valid():
            # LOGIN HERE
            login(request, form.get_user())
            return redirect('myapp:home')
    else:
        form = AuthenticationForm()

    return render(request, 'login_templage.html', {"form": form})
```

Logout

Custom logout logic.

```
from django.contrib.auth import login, logout

def logout(request):
    if request.method == "POST":
        logout(request)
        return redirect("main:home")
    else:
        form = ...

    return render(request, 'custom_auth/logout.html, {"form": form}
```

▼ Authorization

Key thing to know:

Process of granting permissions to an Authenticated user or device. Django, automatically, creates a list of available permissions (see Django admin.), these permissions are stored at `auth_permission` table for users permissions and `auth_group_permissions` for Group permissions. Permissions are assigned to each user at the `auth_user_user_permissions` table [1:M].

▼ Granting users permissions

User Permissions: Django provides a flexible system for defining and assigning permissions to users. Permissions can be assigned directly to individual users or to groups. These permissions control what actions users can perform within the application, such as creating, reading, updating, or deleting specific types of data.

▼ View methods (URLs) user access.

Django can control access to certain views (function based and class based views). It is done mainly via the `login_required` decorator from `django.contrib.auth.decorators`.This decorator can be used as decorator for views functions and classes in the `views.py` file. And, as a method in the `urls.py` file.

Examples:

```
from django.urls import path
from django.contrib.auth.decorators import login_required
from . import views

urlpatterns = [
    path('create', login_required(views.create), name='create'),
    path('<str:slug>', login_required(views.index), name='index'),
    path('', login_required(views.home), name='home'),
]
```

```
from django.contrib.auth.decorators import login_required


@login_required
def home_view(request)
    return HTTPResponse('<h1>Home Page</h1>')
```

The `login_required` decorator can take in several arguments:

1. `login_url: str = 'app/login/'`-Redirect non logged in user to the log in page.

Class based view

With a class based view you need to user the `LoginRequiredMixin` class, instead f the `login_required` decorator.

The `LoginRequiredMixin` is designed specifically for class-based views and ensures that the user is authenticated before they can access the view.

How to Use `LoginRequiredMixin`

Here's a step-by-step guide to using the `LoginRequiredMixin` with a class-based view:

1. **Import the `LoginRequiredMixin`:** You need to import `LoginRequiredMixin` from `django.contrib.auth.mixins`.
2. **Add `LoginRequiredMixin` to Your View:** The `LoginRequiredMixin` should be the ***first mixin*** in the inheritance chain of your class-based view to ensure that the login requirement is checked before anything else.

Example:

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import TemplateView

class ProtectedView(LoginRequiredMixin, TemplateView):
    template_name = 'protected_page.html'

    # Optionally, you can set a custom `login_url` for this view
    login_url = '/custom_login/'

    # Optionally, you can specify a redirect field name (default is
    redirect_field_name = 'redirect_to'
```

▼ HTML access in Templates

▼ Revoking users permissions (banning)

▼ Adding custom permissions

You can create a custom permission object and add it to a specific user(s) or group(s).

▼ Group management

Group Management: Django allows you to create and manage user groups. Groups are a way to categorize users and assign permissions to multiple users at once. This simplifies the process of managing permissions for large numbers of users with similar access needs.

In order to manage users groups and permissions the superuser is required. To create one user `python3 manage.p y createsuperuser`.

▼ Group Creation

Each user present in a group will automatically share permissions that were chosen for the group.

Via Django admin panel

1. Add a new group in Django Admin panel, name it.
2. Choose permissions for this group.

Via Django shell

1. Open Django shell: `python3 manage,py shell` .
2. Import required modules:
    a. `from django.contrib.auth.models import Group, Permission, User`

b. `from django.contrib.contenttypes.models import ContentType` .

3. Create a new Group:
   a. `mod, created = Group.object.get_or_create(name="mod")` Name can be any.
4. After creating the group add permissions to it:
   a. Find appropriate permissions:
      i. Get a model content type, so you could find permissions related to a particular model:

      `ct = ContentType.object.get_for_model(model=Poster)` .

      ii. Get a Query-Set of permission for the model: `perms = Permission.object.filter(content_type=ct)` .
   b. Add permissions to the group: `mod.Permissions.add(*perms)` ,

      you can either unpack all avail permissions or add them one by one.
5. Add a User to the Group:
   a. Get a user: `user = User.object.filter(username="sergei").first()`.
   b. Add the user to the group: `mod.user_set.add(user)`.

▼ Check Groups within SQL query

```
SELECT
    auth_group.name AS groupname,
    auth_user.username,
    auth_user.is_staff
FROM auth_user
LEFT JOIN auth_user_groups ON auth_user.id = auth_user_groups.user_id
LEFT JOIN auth_group ON auth_group.id = auth_group.id;
```

▼ Management commands

▼ Creating custom commands

▼ Signals

▼ Table of built-in signals

▼ Internalization & Localization (i18h and l10n)

Internalization

The process of preparing your application for localization, i.e., making it able to support multiple languages.

Localization

Adapting your application for different languages, regions, or cultures (e.g., translating text, formatting dates, numbers, etc.).

▼ Settings

1. Localization settings: (Note. can user `gettext` to translate name of the languages.)

```
LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'UTC'
USE_I18N = True  # strings translation.
USE_L10N = True  # local time, data and number formats.
USE_TZ = True  # Timezone support.
LANGUAGES = [
    ('en', 'English'),
    ('ru', 'Russian')
]
LOCALE_PATHS = [os.path.join(BASE_DIR, 'locale')]
```

2. Middleware set up. Need to be set in a specific place, after `session` Middleware and before `common` Middleware that is responsible for URL routing.

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',  # Localization midd
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    # Custom. Avail for all project apps.
    'posters.middleware.EnsureSessionKeyMiddleware',
]
```

3. Set Up URLs

```
from django.contrib import admin
from django.conf.urls.static import static
from django.urls import path, include, re_path
from django.conf import settings
import debug_toolbar
# NOTE: Add in production!
from django.conf.urls.static import static
from django.views.static import serve
from django.conf.urls.i18n import i18n_patterns


urlpatterns = [
    # NOTE: Uncomment in production!
    # re_path(r'^media/(?P<path>.*)$', serve, {'document_root': setti
```

```
    # re_path(r'^static/(?P<path>.*)$', serve, {'document_root': sett
    path('admin/', admin.site.urls),
    path('i18n/', include('django.conf.urls.i18n')),
]


urlpatterns += i18n_patterns (
    path('posters/', include(('posters_app.urls', 'posters_app'), nam
    path('user_account/', include(('user_account_app.urls', 'user_acc
)

# NOTE: Comment in Production!
if settings.DEBUG:
    urlpatterns += [path('__debug__/', include(debug_toolbar.urls))]
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.
```

▼ Workflow

1. In templates:
    a. Load tag: `{% load i18n %}`.
    b. Use `{% trans "this text will be translated" %}` in templates. Example:

```
{% load i18n %}
<p>{% trans "Welcome to my website!" %}</p>
```

2. In views and etc.
    a. import `from django.utils.translation import gettext as _`.
    b. Use like in the example:

```
def my_view(request):
    message = _("Welcome to my website!")
    return render(request, 'my_template.html', {'message': message
```

3. Create the `locale` directory in the project root or each app individually.
4. Make locale files: `python3 manage.py makemessages --all` .or `-l ru`. For a specific language.
5. Go to `locale` directoty and write translation to each string.
6. Compile translations: `python3 manage.py compilemessages`.

▼ Language select switch

1. Load tag in the template: `{% load i18n %}`.
2. Example:

```
<li class="nav-item dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="
    <ul class="dropdown-menu" aria-labelledby="navbarDropdown">
```

```
  {% get_current_language as LANGUAGE_CODE %}
  {% get_available_languages as LANGUAGES %}
  {% get_language_info_list for LANGUAGES as languages %}
  {% for lang in languages %}
  <li>
      <a class="dropdown-item" href="/{{ lang.code }}{{ request.get_f
      {{ lang.name_local }}
      </a>
    </li>
    {% endfor %}
    </ul>
</li>
```

General knowledge:

▼ Components

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It is composed of several components that work together to provide a full-featured web development experience. Here is a list of the core components of the Django framework:

▼ Components

1. Models:
    a. Description: Models define the structure of data in each application, and provide a way to manage it through Django ORM.
    b. Key features:
        i. Table (model) creation. Supports custom fields and functionality. Like Image path building using `Pillow` module and data validation using validators.
        ii. Handle database migrations.
        iii. Query Database using high-level API.
2. Views:
    a. Description: View handle the logic of an application. They accept and return data to the user. They can fetch data from the db, apply business logic, and pass data to templates.
    b. Key features:
        i. Function-Based Views (FBVs): Single function can handle requests and send responses.
        ii. Class-Based Views (CBVs): Offering a more modular and reusable approach by Using Python classes to encapsulate some logic.
3. Templates:
    a. Description: Django Templates Language can create dynamic web pages.

b. Key features:

i. Tags. `{% for value in QuerySet %}` etc.

ii. Filters. `{{ value|lower }}` etc.

iii. Template inheritance to reuse common elements. `{% extends from 'base.html' %}` .

4. URLs and Routing:

a. Description: Django URLs dispatcher allows to differ URL patterns that map to views, enabling clean and organized URLs structure.

b. Key features:

i. URLs defined using Django functions `path()` and `re_path()`.

ii. Support name URL patterns for easy reverse URL look ups.

iii. Middleware integrating for processing requests and responses globally. E.g. ‘Ensure session_key exists’ Middleware.

5. Forms:

a. Description: Django powerful form handling library to process and validate user inpt data.

b. Key features:

i. Form classes to define, validation and form rendering.

ii. Custom validating and error handling.

6. Admin interface:

a. Description: built-in interface for managing applications data and models.

b. Key features:

i. Auto generate CRUD interface.

ii. User authentication and permission management.

7. Authentication and Authorization:

a. Description: Built-in authentication system for managing users (Anonymous as well), passwords and permissions.

b. Key features:

i. Built-in views for logging, logout, password reset, etc.

ii. User models and groups for managing users roles and permissions.

iii. Custom user modes for extending default user functionality.

8. Middleware:

a. Description: Middleware is the way to process requests before they reach views or after the view has processed.

b. Key feature:

i. Built-in Middleware for security, session management authentication, and more.

    ii. Ability to make custom Middleware. Useful for logging, request modification, headers modification, checking views and models for the ability to accept requests.

9. Internalization and Localization (i18n and l10n):
    a. Description: Adds language support and formats.
    b. Key feature:
        i. Built-in translation system for text within templates and code.
        ii. Local Middleware to track user preferences and adjust content for them.
        iii. Formatting dates, time, etc.
10. Testing:
    a. Description: Django testing framework based on `Unittest` library.
    b. Key features:
        i. Unittesting for model, forms, view and other components.
        ii. Client test by simulating request and checking responses.
        iii. Database testing for data integrity.
11. Security features:
    a. Built-in security feature to protect against common web vulnerabilities.
    b. Key Features:
        i. Cross-Site request Forgery (CSRF) protection.
        ii. Cross-Site Scripting (XSS) protection.
        iii. SQL injection protection.
        iv. Clickjacking protection with the X-Frame-Options Middleware.
12. Django Rest Framework:
    a. Description: Although not a part of the core Django framework, Django Rest Framework is a powerful and flexible toolkit for building Web APIs.
    b. Key features:
        i. Serializes for converting complex data type to and from JSON.
        ii. Class-based views for handling HTTP methods.
        iii. Authentication and permission classes for API security.
13. Django Channels:
    a. Description: Django Channels extends Django functionality to handle WebSockets, allowing real-time communication and asynchronous task handling.
    b. Key Features:
        i. Support WebSockets and background tasks.
        ii. Asynchronous views and consumers for handling long-running processes.
        iii. Integration with Django ORM and Middleware.

14. Caching:
    a. Description: Django provides various caching mechanisms.
    b. Key feature:
        i. In-memory caching, file-based caching, and data caching.
        ii. Cache Framework for setting up different cache backends.
        iii. Decorators and templates tags for caching views and template fragments.
15. Signals:
    a. Description: Signals allow decoupled applications to get notified when certain actions occur elsewhere in the application.
    b. Key features:
        i. Built-in signals for common events (e.g., `post_save`, `pre_delete` .).
        ii. Ability to make custom signals.
        iii. Receiver functions to handle signals.
16. Management Commands:
    a. Description: Set of management command to perform different task.
    b. Key Features:
        i. Built-in commands: `runserver`, `makemigrations`, etc.
        ii. Ability to make custom management commands.
        iii. Extensible command system using Django's management API.

▼ Django Module Structure Overview

Django's source code is organized into a collection of packages and modules, each responsible for different parts of the framework

▼ Models structure

1. **django.conf**
    - **Description:** Provides Django's configuration framework and handles project settings.
    - **Key Modules:**
        - `django.conf.settings`: Access and manage Django settings.
        - `django.conf.urls`: Tools for URL routing, including `url()` and `include()`.
2. **django.core**
    - **Description:** Contains the core components of Django, including utilities, management commands, and exceptions.
    - **Key Modules:**
        - `django.core.exceptions`: Common exceptions used throughout Django (e.g., `ValidationError`, `ObjectDoesNotExist`).
        - `django.core.handlers`: Request and response handling.

- `django.core.management`: Command-line utilities (e.g., `startproject`, `makemigrations`).
- `django.core.mail`: Email utilities for sending emails.
- `django.core.cache`: Caching framework for managing cache backends.
- `django.core.serializers`: Tools for serializing and deserializing data (e.g., JSON, XML).

3. **django.db**
   - **Description:** Provides Django's database abstraction layer, including ORM, models, and query APIs.
   - **Key Modules:**
     - `django.db.models`: Core module for defining models and fields.
     - `django.db.models.query`: QuerySet classes and related utilities.
     - `django.db.backends`: Database backends for various databases (e.g., PostgreSQL, MySQL, SQLite).
     - `django.db.migrations`: Tools for managing database schema changes through migrations.
4. **django.http**
   - **Description:** Contains classes and functions for handling HTTP requests and responses.
   - **Key Modules:**
     - `django.http.HttpRequest`: Represents an HTTP request.
     - `django.http.HttpResponse`: Represents an HTTP response.
     - `django.http.JsonResponse`: A subclass of `HttpResponse` for JSON data.
     - `django.http.Http404`: Exception raised for a "Page Not Found" error.
5. **django.shortcuts**
   - **Description:** Provides helper functions to reduce the amount of code needed for common operations.
   - **Key Functions:**
     - `render()`: Combines a template with a context dictionary and returns an `HttpResponse` object.
     - `redirect()`: Returns an `HttpResponseRedirect` to the given URL.
     - `get_object_or_404()`: Retrieves an object or raises a `Http404` exception if not found.
     - `get_list_or_404()`: Retrieves a list of objects or raises a `Http404` exception if none are found.
6. **django.template**
   - **Description:** The template system, which allows for rendering HTML and other text-based formats.
   - **Key Modules:**

- `django.template.loader`: Functions for loading templates.
- `django.template.backends`: Interfaces for different template engines.
- `django.template.context`: Context classes for templates.
- `django.template.defaulttags`: Built-in template tags.
- `django.template.defaultfilters`: Built-in template filters.

7. **django.urls**
   - **Description:** Tools for URL routing and handling.
   - **Key Modules:**
     - `django.urls.path`: Function to define URL patterns using the newer path converters.
     - `django.urls.re_path`: Function to define URL patterns using regular expressions.
     - `django.urls.include`: Function to include other URL configurations.
     - `django.urls.reverse`: Function to reverse resolve URLs.
8. **django.utils**
   - **Description:** A collection of utility functions and classes for various tasks.
   - **Key Modules:**
     - `django.utils.timezone`: Time zone-related utilities and classes.
     - `django.utils.translation`: Tools for internationalization and localization.
     - `django.utils.decorators`: Common decorators used in Django views.
     - `django.utils.datastructures`: Data structures like `MultiValueDict`.
9. **django.forms**
   - **Description:** The form handling framework, which allows for creating and validating forms.
   - **Key Modules:**
     - `django.forms.models`: ModelForm classes that map models to forms.
     - `django.forms.fields`: Form field classes (e.g., `CharField`, `EmailField`).
     - `django.forms.widgets`: Form widget classes (e.g., `TextInput`, `Select`).
     - `django.forms.formsets`: Tools for handling formsets, groups of forms.
10. **django.contrib**
    - **Description:** A collection of optional Django applications that provide common functionalities.
    - **Key Submodules:**
      - `django.contrib.admin`: The Django admin interface for managing models.
      - `django.contrib.auth`: The authentication system, including user models and permissions.
      - `django.contrib.sessions`: Session management for tracking user sessions.

- `django.contrib.sites`: Framework for managing multiple sites with one Django installation.
- `django.contrib.staticfiles`: Tools for managing static files (e.g., CSS, JavaScript).

11. **django.middleware**
    - **Description:** Middleware is a way to process requests globally before they reach the view or after the view has processed the request.
    - **Key Modules:**
        - `django.middleware.csrf.CsrfViewMiddleware`: Provides Cross-Site Request Forgery protection.
        - `django.middleware.common.CommonMiddleware`: Performs various common operations, like URL rewriting and ETag handling.
        - `django.middleware.security.SecurityMiddleware`: Provides various security enhancements, such as setting HTTP headers.
12. **django.test**
    - **Description:** Provides tools and classes for testing Django applications.
    - **Key Modules:**
        - `django.test.TestCase`: A subclass of Python's `unittest.TestCase` that supports Django-specific features.
        - `django.test.Client`: A test client that acts like a browser for testing views and URLs.
        - `django.test.SimpleTestCase`: A subclass for tests that do not require database access.